Why Am I Excited About WebAssembly?

Jul 17, 2022

I expand on these ideas in my S4 presentation WebAssembly at the IoT Edge and in my Reactive Summit presentation WebAssembly for a Reactive Internet of Things.

WebAssembly, commonly abbreviated as Wasm, is a portable binary intermediate language that can execute on different platforms in a virtual machine. The intermediate language is compiled, either just-in-time (JIT) or ahead-of-time (AOT), to the machine code appropriate for the architecture. Many programming languages compile to WebAssembly, including C, C++, C#, Rust, Go, and Swift, similar to how Java, Scala, and Clojure compile to Java bytecode and execute in the Java Virtual Machine, and C#, F#, and Visual Basic compile to Common Intermediate Language (CIL) and execute in the Common Language Runtime (CLR).

WebAssembly was first released in 2017 and is supported by all popular web browsers. It was originally developed to enable high-performance applications on client webpages. Efficient code for manipulating images or video could be written in a language other than JavaScript, for example, C++, yet still execute in the browser. It also allowed legacy code with years of investment to be ported to a web browser. Google Earth, Adobe Photoshop, and Autodesk AutoCAD are prominent examples.[1]

While tremendously valuable, using WebAssembly in a web browser is not what excites me. In fact, I know very little about web development. What excites me is the potential to securely and efficiently run code at both the edge and in the cloud to support the Internet of Things (IoT).

The Edge

When I refer to IoT, I am referring to devices that interface with, collect data from, and control physical things, like thermostats, solar inverters, factory equipment, medical devices, or connected cars.[2] IoT computing platforms are a mix of firmware and software and span everything from:

- Dedicated firmware on a microcontroller.
- Embedded systems with lightweight operating systems that provide code reuse and abstraction (e.g., real-time computation, memory management, networking), while respecting resource and power constraints.
- Industrial computing platforms that run on general-purpose microprocessors with abundant processing, memory, storage, and power, and a full Linux operating system.

Over-the-air updates for firmware and software are critical for improving IoT products over time, and applying security patches, but present a host of challenges. If the device performs safety-critical functions, like actuating

challenges. If the device performs safety-critical functions, like actuating power, medical equipment, or a connected car, the firmware and software cannot be updated without rigorous testing. If the device requires certification, updating it may require recertification. Updates are disruptive and need to be scheduled when the device is not performing its critical functions. Even just performing the update presents risks: if the update ends up bricking millions of devices and requires manual intervention, it could prove an existential risk to the business or the safety of individuals.[3]

As a result, edge firmware and software typically have long release cycles with significant hardware-in-the-loop testing. Firmware and software are usually bundled together and are only updated on the order of weeks or months. There is a natural reluctance to update any code at run-time given the potential for bugs and remote code execution (RCE) vulnerabilities. This makes iterating on product improvements, or quickly providing security patches, difficult.

A firmware component that provides critical functions must always have a rigorous testing and release process. However, for a software component, it would provide a huge competitive advantage if it could be updated flexibly, even on-demand. Imagine the potential to iterate on a machine learning model daily, rather than every few months.

This is why I find both the execution model and the security model of WebAssembly exciting. WebAssembly binaries can be very small, sometimes on the order of bytes, and they are portable across platforms. In terms of security, WebAssembly executes in a sandbox and the basic instruction set provides for nothing more than pure functions—numerical operations on its own memory. The WebAssembly code cannot read from, or write to, the file system, open ports, or access shared memory. The WebAssembly code can take advantage of functions or variables provided by the host, but only if the host provides these explicitly through imports. In other words, the host is in full control of what the WebAssembly code can do. The host can even limit

the memory used or meter the number of instructions executed by the WebAssembly code. For building more complex applications that need to open ports or perform IO, there is the WebAssembly System Interface (WASI) which defines a standard set of system imports while still providing sandboxed execution. However, as I will describe more below, I am most interested in WebAssembly for pure functions with no side effects.

Considering a concrete example will help. Consider an electric water heater that responds to the cost of electricity and avoids heating when power is the most expensive, while still predicting and respecting the customer's typical usage. The business logic that determines the control setpoint might be a simple state machine or an intensive numerical computation that solves an optimization problem. In either case, it is a pure function with no side effects. The software running on the water heater gathers the input data (e.g., price forecast, historical usage, time of day) and performs all necessary IO. It passes the inputs to the setpoint function and, based on the output, performs the necessary actuation.

What if you want to modify the setpoint function to improve the optimization? If the software and the firmware are bundled together, there is no way to update it without going through a full testing and release cycle. But if the setpoint function was a WebAssembly module, it could be updated flexibly, without a new firmware release. The code can be signed so we know it is from a trusted source. We also know the code cannot do anything malicious, like modify host memory or write to the file system, because it is sandboxed. Some might be concerned about the WebAssembly code providing a bad setpoint, either maliciously or through a programming error, but the safety critical firmware must protect against this already, even if the software cannot be updated at run-time. The distinction is that the setpoint function is only providing the setpoint—the firmware must still ensure the setpoint is within acceptable bounds that may be determined by physical or regulatory constraints.

When I share my excitement for this approach, most people are sceptical of WebAssembly and suggest alternatives. Unquestionably, WebAssembly is not required to update software at run-time—one could download a standalone binary or a Docker container, verify it is signed and from a trusted source, and sandbox its execution through systemd or the configuration of the container run-time. However, as demonstrated by the SolarWinds SUNBURST attack, supply-chain attacks can render even trusted code vulnerable. WebAssembly should be far safer, especially for pure functions, because its instruction set is so limited, and its security model is restrictive rather than opt-in.[4] Setting aside the security challenges, if you need to support heterogeneous devices, the binaries would need to be compiled for the target architecture and there is no portability. Binaries and containers can also be large, on the order of megabytes, and downloading them can be slow and incur significant transmission costs. Finally, running a binary or a container to execute a pure function has significant penalties for resource consumption and start-up time.[5] In the limit, trying to address all of these issues—small binaries, fast execution, portability, strict sandboxing, metering execution—feels like reinventing WebAssembly.

Why I am excited about WebAssembly for the IoT edge:

- It has small, portable binaries and near-native execution for many workloads.
- It has a sandboxed security model where the host is in full control, and the basic instruction set is limited to pure functions.
- When used for pure functions expressing business logic or numerical computation, it provides a means to securely and reliably update this code at run-time without a firmware release.
- Being able to flexibly update run-time code outside of lengthy firmware development, testing, and release cycles becomes a strategic product differentiator.

The Cloud

The adoption of WebAssembly in the cloud is limited, but it is more mature than the IoT edge. Both Cloudflare and Fastly adopted WebAssembly to run client code and move it closer to the customer through their global networks. Redpanda, a messaging infrastructure compatible with the Kafka API, also adopted WebAssembly to run client code in event streams. For these vendors, WebAssembly is attractive because it is portable and sandboxed. Customers can write code in any language, as long as it compiles to WebAssembly, and because the vendor is running untrusted code, often in a multi-tenant environment, the strict sandboxing protects the vendor against malicious or buggy code, and helps provide tenant isolation. Cloudflare and Redpanda both use V8, the high-performance JavaScript and WebAssembly engine used in Google Chrome and Node.js, whereas Fastly built their own compiler and run-time called Lucent.[6]

In my work, I am not as concerned with running untrusted code because the software is vertically integrated from firmware all the way to cloud services. I am excited about WebAssembly in the cloud for two reasons: 1) moving code from the cloud to the edge for interoperability, fast iteration, and cost savings, and 2) abstracting the cloud run-time from the developer for productivity, portability, and security.

The ability to move computation from the cloud to the edge is revolutionary. Computation can be developed, tested, and benchmarked in the cloud using historical data stored in the cloud, simulated data, or live data streamed or queried from the IoT device. When you are satisfied with the results, the code can be moved from the cloud to the edge.[7] WebAssembly binaries are small, so they can be stored as configuration and synchronized to the IoT devices using existing mechanisms for desired-state configuration management. This makes it possible to iterate on edge computation, reliably and securely, in hours or days, rather than weeks or months. Even experimental

computation, for example, a new machine learning model, can run at the edge in shadow mode and report results back to the cloud for evaluation. In addition to this flexibility, moving computation to the edge can provide significant costs savings by taking advantage of existing edge computation, rather than consuming additional cloud resources.[8]

I have extensive experience with Akka, a toolkit for actor-model programming, distributed computing, and streaming data. Akka has been a wonderful platform for building reliable and scalable services for IoT. With Akka, the run-time is abstracted from the developer. The developer writes pure functions that are then composed into actors or streams. For actors, the run-time handles concurrency, distribution, location transparency, and failure. For streaming, the run-time handles the dynamics through asynchronous backpressure and the Reactive Streams standard. The details are too involved to describe here, but I have written about this in the past, including how Akka enables digital twins for IoT devices through lightweight actors.[9] Despite these great abstractions, after developing a large number of services, updating the run-time becomes a significant burden, because the run-time itself is compiled into application binaries. This is certainly not unique to Akka. Most applications that have a simple HTTP or a gRPC interface have the server compiled into the application binary, when all the developer is really concerned about is writing the business logic associated with each route. As a result, developers end up maintaining a lot of boilerplate and framework code.

To demonstrate the challenge, consider hundreds of services using an HTTP server compiled into the application binary. There are likely many different versions depending on when the application was last compiled and deployed. If there is a security vulnerably identified in a specific version of the HTTP server, what services need to be patched and where are they running? This is not an easy question to answer.[10] Most people attack this problem with tooling, but what if the run-time could be updated independently from the

business logic? This would be similar to how host operating systems can be patched underneath Kubernetes and no one needs to care, as long as the pods are rescheduled on another host.

It would be great to have a high-level run-time like Akka but abstracted entirely outside of the code the developer writes. You can think of this as a Functions as a Service (FaaS) model, but with even less boilerplate and richer primitives for handling messaging, dynamics, failure, and state. WebAssembly holds a lot of promise in being able to separate the run-time environment from the business logic and there are a few platforms evolving in this direction.[11]

Krustlet is a means of running WebAssembly on Kubernetes. Because WebAssembly has small binaries that can be downloaded quickly and run efficiently without a lot of start-up overhead, it is attractive for serverless functions.[12] Lunatic is a runtime inspired by Erlang/OTP for running WebAssembly actors for massively parallel workloads. Suborbital is a platform for declaratively composing WebAssembly modules into applications and has support for managing state and interfacing with streaming data systems. wasmCloud can run actors and it also has capability providers for messaging and interfacing with relational databases and key-values stores.[13] wasmCloud looks the most like what I am familiar with with Akka, where it is easy to model IoT devices with actors. One thing I would not want to lose with any of these models is type safety and there needs to be system-wide type checking that can be performed statically, not just at run-time.[14]

Why I am excited about WebAssembly for the IoT cloud:

- Computation can be developed in the cloud, then flexibly and securely moved from the cloud to the edge.
- Code can be managed alongside existing desired-state configuration for the IoT devices. In other words, code as configuration.

- Abstracting more of the run-time from the business logic should make operating cloud services easier over time, continuing the trend of abstracting the operating system, the network, and service orchestration further up the stack.
- The low overhead is excellent for serverless functions.

Challenges

It is still early days for WebAssembly on the edge and in the cloud. The biggest challenge may be picking a WebAssembly run-time. For the edge, Wasmer, Wasmtime, Wasm3, WasmEdge, and others, all seem like viable options that can be used as a library from various programming languages. WebAssembly is portable across run-times, but changing from one host run-time to another would be a major investment once a significant amount of code or tooling has been developed. If WebAssembly is only used for pure functions, however, porting code from one host run-time to another may be relatively straightforward. While such active development is exciting, it will be beneficial for the industry to consolidate around a small number of ubiquitous run-times. Some people will be concerned about the experience of debugging and troubleshooting, but I think this is of minor concern when using WebAssembly for pure functions.

Selecting a cloud run-time will depend largely on the type of workload you want to run. For example, do you want to run pure functions, entities that need to maintain state (e.g., digital twins), services that need to interact with databases or streaming data, or do you want to evaluate your WebAssembly code for the edge running in the same host run-time? As the cloud ecosystem for WebAssembly becomes richer, as long as your code is WebAssembly and WASI compliant, it should be portable to another run-time. However, for the cloud to abstract interacting with databases, key-value stores, message buses, etc., in a portable way, WASI, or another standard built on top of WASI,

needs to include these interfaces. WASI is important to enable many applications, but it might be the wrong layer of abstraction for most developers.[15]

Part of the appeal of WebAssembly is that it can be targeted by many different programming languages. Legacy code can also be ported to WebAssembly. If the legacy code is C or C++, WebAssembly is arguably a safer environment for execution given the limited instruction set and stack-based virtual machine. Python, however, feels awkward. If my understanding is correct, since Python is an interpreted language, the interpreter must also be included in the WebAssembly code. Given a choice, I would much prefer to program WebAssembly from a language that is strongly typed, supports functional programming, affords compile-time memory safety, and has great tooling. Rust seems to be by far the most attractive language for any new development.

Finally, if your application must take advantage of multiple cores, support for multi-threading in library run-times is still new and limited. That said, for many applications, like numerical computation, WebAssembly is very fast and runs with native performance. The performance is also very predictable since there is no garbage collection. Most run-times also have support for Single Instruction, Multiple Data (SIMD) which use the processor more efficiently for repetitive calculations on large data sets. For the majority of the applications I am looking at, multi-threading should not be a major limitation, and WebAssembly should deliver excellent and efficient performance.

Conclusion

WebAssembly is portable, can be targeted from multiple programming languages, has small binaries, runs efficiently, and has a very strict sandboxing model. This makes it exciting for IoT:

- At the edge, it can enable flexibly moving code from the cloud to the edge without requiring a firmware update.
- In the cloud, it can abstract the run-time from the business logic and allow it to become part of the infrastructure.

Why am I excited about WebAssembly? I have not used WebAssembly in a product, so I am not speaking from direct experience, but I am excited about WebAssembly for IoT, for both the edge and the cloud, and for a more flexible relationship between the two. As it continues to evolve, WebAssembly has the potential to be the platform executing an enormous amount of IoT computation. In a subsequent essay, I will provide a concrete example that I hope will make these points easier to digest.

---

1. The first time I heard of embedding C++ on the web was a talk on Emscripten and asm.js at CppCon 2014. ↩
2. For more perspectives on IoT, including an expanded definition, see my essay and presentation The State of the Art for IoT. ↩
3. Imagine having to send people to locations all over the world to recover thousands of devices by hand. It could take weeks or months depending on the accessibility of the devices and the availability of trained staff. While the devices are not functioning, lives could be at risk. ↩
4. The run-time is still susceptible to security vulnerabilities, but this is also true of operating systems and container run-times. ↩
5. Cold start time for a container or a standard serverless function can be hundreds of milliseconds or more, whereas a WebAssembly function can startup orders of magnitude faster. See A lightweight design for serverless Function-as-a- Service and How Compute@Edge is tackling the most frustrating aspects of serverless. ↩
6. Note, Lucent is end-of-life and Fastly is investing in Wasmtime. ↩
7. Microsoft Azure has supported Azure Stream Analytics for some time with the ability to run the same streaming calculation in the cloud or on the edge. I assume

the portability of Azure Streaming Analytics is leveraging .NET and the CLR in the cloud and at the edge. ↩

8. Some people will be concerned about keeping track of the versions of the WebAssbmely binaries that each device is running as well as the combinatorial testing for the interactions among versions. This is a challenge, but I do not see it as any different than the management or testing required for devices with a large number of configuration settings, which is typical for industrial IoT devices. ↩
9. Refer to my articles and associated talks: Essential Software Tools for Developing Operational Technologies, On Eliminating Error in Distributed Software Systems, and The State of the Art for IoT. For a discussion of digital twins, see the talk the Tesla Virtual Power Plant. ↩
10. I expand on some of the challenges of operating, maintaining, and evolving a large number of microservices in my article Concordance for Microservices. ↩
11. People familiar with Enterprise JavaBeans (EJB) get apprehensive about these approaches. I have never used this technology, but I am sure there are lots of lessons to be learned. ↩
12. While the platforms for serverless functions from the major cloud vendors do not yet natively support WebAssembly, I suspect it will become standard over time. It may even offer portability across service providers. ↩
13. It relies on NATS for messaging, and is compatible with NATS Global Service, an amazing technology. ↩
14. Kalix does not use WebAssembly, but it does abstract most of the run-time into the service provider and uses Protocol Buffers for type safety across service boundaries. ↩
15. It reminds me some of the Reactive Streams standard and people adopting it directly, rather than library developers using it for higher-level abstractions. See Akka Streams: A Motivating Example. ↩